

Multi GPU Performance of Conjugate Gradient Solver with Staggered Fermions in Mixed Precision

Yong-Chull Jang*, Hyung-Jin Kim, and Weonjong Lee

Lattice Gauge Theory Research Center, FPRD, and CTP

Department of Physics and Astronomy, Seoul National University, Seoul, 151-747, South Korea

E-mail: wlee@snu.ac.kr

GPU has a significantly higher performance in single-precision computing than that of double precision. Hence, it is important to take a maximal advantage of the single precision in the CG inverter, using the mixed precision method. We have implemented mixed precision algorithm to our multi GPU conjugate gradient solver. The single precision calculation use half of the memory that is used by the double precision calculation, which allows twice faster data transfer in memory I/O. In addition, the speed of floating point calculations is 8 times faster in single precision than in double precision. The overall performance of our CUDA code for CG is 145 giga flops per GPU (GTX480), which does not include the infiniband network communication. If we include the infiniband communication, the overall performance is 36 giga flops per GPU (GTX480).

The XXIX International Symposium on Lattice Field Theory - Lattice 2011

July 10-16, 2011

Squaw Valley, Lake Tahoe, California

*Speaker.

1. Introduction

CPU has been improving its computing performance but does not yet quench the thirst of those demanding users who need more computing power for their numerical challenges such as lattice QCD. Graphic processing units (GPU) opens a new era for high performance computing. GPU is originally designed to handle 3-dimensional graphic images. To achieve extremely high performance with geometric data, GPU is designed of simple and tiny processors. More modules are used for the data processing and, not for the data cache, nor for the flow control. Hence, GPU are very different from typical CPUs by construction. GPUs are very appropriate for highly intensive and parallelized scientific computation. At the beginning, programming GPU was quite challenging and difficult. Recently, Nvidia has introduced the CUDA library, which allow the users to program the code for GPU easily.

Since then, there have been several ways to program the GPU code: the Nvidia CUDA, Open Graphic Library (Open GL), and Open Computing Language (Open CL) APIs. In this paper, we focus on CUDA and its applications. The CUDA provides us a user-friendly programming environment based on the C, C++ programming language for GPU. All of our CPS library codes are compiled and tested in CUDA version 3.2 and compute capability 1.3 mode. We made the CUDA version of CG subroutines that were implemented as a part of the Columbia Physics System (CPS) library.

2. Conjugate gradient method

Conjugate gradient (CG) algorithm [7] is an iterative method for solving a linear algebraic equation of the following form.

$$\mathbf{b} = A\mathbf{x}, \quad (2.1)$$

where A is a $n \times n$ positive definite Hermitian matrix. \mathbf{x} and \mathbf{b} are complex vectors in the n dimensional space. Matrix A and vector \mathbf{b} are given and \mathbf{x} is a solution vector that we want to obtain. Using the CG method we can get the solution \mathbf{x} up to the numerical precision what we want to achieve.

In Fig. 1, we show the structure of CG algorithm. In the CG sequence, we have a number of linear algebra equations such as vector addition, dot product, and scalar multiplication and so on. All of these linear algebra operations are implemented using CUDA library. Because most of these operations are not dominant part in CG operation, there is no special applied optimization for those functions except Dirac operation.

In Fig. 1, $A\mathbf{d}$ and $A\mathbf{x}$ are Dirac operations with staggered fermions [8]. Basically, the Dirac operation is a matrix-vector multiplication. This is the most dominant part in CG sequence. The matrix A is defined as follows.

$$\mathbf{h} = A\chi \quad (2.2)$$

$$A = -D^2 + m^2 \quad (m \text{ is quark mass}) \quad (2.3)$$

$$D_{x,y} = U_\mu(x)\delta_{y,x+\mu} - U_\mu^\dagger(x-\mu)\delta_{y,x-\mu} \quad (2.4)$$

$$D\chi(x) = \sum_\mu U_\mu(x)\chi(x+\mu) - U_\mu^\dagger(x-\mu)\chi(x-\mu) \quad (2.5)$$

$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$	\mathbf{r} : residual vector
$\mathbf{d} = \mathbf{r}$	\mathbf{d} : directional vector
$\delta_{new} = \mathbf{r}^\dagger \mathbf{r}$	ε : tolerance
$\delta_0 = \delta_{new}$	
for($i = 0; i < N_{dim} \& \delta_{new} > \varepsilon^2 \delta_0; ++i$) {	
$\alpha = \delta_{new} / \mathbf{d}^\dagger \mathbf{A} \mathbf{d}$	
$\mathbf{x} = \mathbf{x} + \alpha \mathbf{d}, \quad \mathbf{r} = \mathbf{r} - \alpha \mathbf{A} \mathbf{d}$	
$\delta_{old} = \delta_{new}, \quad \delta_{new} = \mathbf{r}^\dagger \mathbf{r}$	
$\beta = \delta_{new} / \delta_{old}, \quad \mathbf{d} = \mathbf{r} + \beta \mathbf{d}$	
}	

Figure 1: Conjugate gradient algorithm

Here, the phase factor $\eta_\mu(x)$ is multiplied in advance to the gauge link U_μ at the gauge link preconditioning part of CPS library. h is a given as a source vector and χ is a staggered fermion field which corresponds to the CG solution.

At one site of the lattice, a single Dirac operation $D\chi(x)$ needs 1584 bytes of data transfer and 576 number of floating point calculations. Let us consider a MILC fine lattice of $28^3 \times 96$. A single Dirac operation $D\chi(x)$ over the entire, even sites of the lattice needs 0.6 billion number of floating point calculations. And it also needs 1.6 giga bytes of data transfer. As a result, when we use GPUs for CG operations, it is easy to find out that the bottle neck is on the data transfer rather than numerical operation.

3. Mixed Precision CG and implementation on CUDA

3.1 Mixed Precision CG

Historically, there exists a significant gap between single precision performance and double precision performance in GPUs. In the case of the Nvidia GTX480 GPU, the single precision calculation runs 8 times faster than the double precision calculation.

How much we can accelerate the program is depend on the ratio of arithmetic operations and data I/O¹. The data I/O access is dominant in CG. It is the main bottle-neck in CG algorithm. Here, we do not include the infiniband network communication in the data I/O. So, by using the single precision data, the main benefit from it is that we can reduce the data traffic by a factor of 2, between GPU processor and GPU device memory.

To improve this performance of the CG program, the mixed precision method has been used. Mixed precision is implemented by iterative refinement algorithm[2]. The main idea of the iterative refinement is using two types of iterative loops to get the true solution value. At first, by using the single precision iteration, we can approach fast to the roughly estimated solution within inner loop tolerance. And next, double precision or more precise iteration can be used to get the more accurate

¹I/O means input and output to the GPU memory.

solution within the outer loop error range. The iterative refinement procedure in CG is illustrated in Fig. 2. In the inner loop, $A\mathbf{y} = \mathbf{r}_k$ is solved by using single precision CG iteration which gives a

```

 $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$             $\mathbf{r}$ : residual vector
while( $\|\mathbf{r}_k\| > \varepsilon \|\mathbf{r}_0\|$ ){
{
    – Inner Loop –
    Solve  $A\mathbf{y} = \mathbf{r}_k$  within  $\varepsilon^{in}$ 
}
 $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{y}$ 
 $\mathbf{r}_{k+1} = \mathbf{b} - A\mathbf{x}_{k+1}$ 
 $k = k+1$ 
}

```

Figure 2: Mixed precision CG algorithm(ε^{in} is tolerance for low precision inner loop, ε means global convergence for overall CG sequence)

approximate solution to the correction terms to the outer loop solution. From this method, we can replace most (99.8%) of slow double precision iterations by fast single precision iterations, while preserving the total number of CG iterations. As a result, the CG algorithm can run about three times faster.

3.2 Corrected mixed precision CG

It is possible to improve the performance of the above mixed precision CG method further. In the CG, there are two iterative loops (inner and outer) that are completely independent of each other. So information such as the residual vector or directional vector is not shared between the inner and outer loops except corrected solution \mathbf{y} . Hence, it is possible to improve the performance of CG by transferring these informations to inner loops as in Refs. [3] Because informations from the previous run of the inner solver can be recycled for solving the next problem in the inner loop. To transfer the information, the initial value of inner loop vectors are set as

$$\mathbf{y}_0 = 0, \quad \mathbf{r}_0^y = \mathbf{r}_k, \quad \mathbf{d}_0^y = \mathbf{r} + \beta \mathbf{d} \quad (3.1)$$

where \mathbf{r}_0^y and \mathbf{d}_0^y are the initial residual vector and the initial directional vector in the inner CG loops, respectively. The main idea of this initial condition is preserving the orthogonal descend direction \mathbf{d}^y from the previous loop. As a result, CG method can continue the iteration in the orthogonal direction, which achieves significantly faster convergence.

The next problem is that using single precision calculation can induce the accumulation of round-off errors. This can cause discrepancy between iterated residual and true residual as the iteration proceed. Furthermore, the solution vector is also vulnerable to this kind of error. To prevent these errors, there are two solutions suggested in the market: one is the reliable update method and the other is the group wise update method [5, 6].

The reliable update method is that if the inner-loop residual is decreased by ε^{in} compared to maximum of all the previous residuals ($\|\mathbf{r}_n^y\| < \varepsilon^{in} \text{Max}(\|\mathbf{r}^y\|)$), it updates the single precision residual vector r_n^y by the double precision residual $\mathbf{r}_{k+1} = \mathbf{b} - A\mathbf{x}_{k+1}$ after we reconstruct the double precision solution vector $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{y}_n$. By using this method, single precision iteration is automatically restarted so that accumulated round off error of residual vector is corrected periodically.

The group wise update method is that if the residual satisfies the condition ($\|\mathbf{r}_n^y\| < \varepsilon^{in} \text{Max}(\|\mathbf{r}^y\|)$), we obtain a double precision solution vector ($\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{y}_n$), recalculate the double precision residual vector, and then reset the single precision solution vector $\mathbf{y}_{n+1} = 0$ while preserving the single precision residual vector as $\mathbf{r}_{n+1}^y = \mathbf{r}_{k+1}$. By updating this way, we can prevent the irregular summation of round-off errors in the solution vector.

We apply both of these methods to our CG code.

All of the inner products are calculated in double precision. After applying these methods, the mixed precision CG runs efficiently and correctly without any overhead due to round-off errors.

3.3 Mixed CG implementation on CUDA

Getting best performance in CUDA programming is a very complicated problem. In our previous paper [1], several optimization methods were applied. Roughly, we apply 4 optimization methods: coalesced memory access, register and shared memory, data compression (SU(3) reconstruction), and extra optimization were used in Ref. [1]. Here, we use the mixed precision method in addition, and adjust the program to accommodate a single precision calculation.

These optimization methods are not the cure-all solutions. The best performance can be achieved by trial and error, when applying these optimization methods. The data size in single precision is half of that in double precision. In CG, the main bottle-neck lies in the data I/O. Naively we expect that the performance will be doubled at least using the mixed precision update.

Using the shared memory has bank conflict problem in double precision calculation [4]. However, the shared memory does not have the same problem in single precision. Since the single precision calculation is dominant (99.8%) in the mixed precision CG, we can use the shared memory as a fast buffer, which lowers the usage of registers. This gives an allowance in high CUDA occupancy rate, which enhances the performance of CG. In practice, we store the η values and position vector in the shared memory. If the number of threads per block is 192, the size of used shared memory is $192 \times 2 \times 4 \times 4\text{bytes} = 6144\text{bytes}$. The register memory is used as a I/O buffer memory. Because the NVCC compiler automatically chooses the variables on the register, it makes the CUDA occupancy low sometimes. In our case, by restricting the number of used registers lower than 42 and with appropriate usage of shared memory, we can achieve better CUDA occupancy(=50%).

We do not use the 8 parameter SU(3) reconstruction [6] for data compression mainly because it causes a big round-off error. Hence, we use 10 parameter SU(3) reconstruction method.

Double precision calculation is the same as explained in our previous paper [1]. Nevertheless, the performance of entire double precision sequences are also improved by adjusting the packet size of data transfer.

In table.1, we present the elapsed time of a single Dirac operator calculation. The data transfer is dominant: 2/3(double precision: DP), and 3/4(single precision: SP). The floating point calculation is sub-dominant: 1/3 (DP) and 1/4 (SP). If we drop out the network communication of MPI

	Double Precision(ms)	Single Precision(ms)
GPU processing time	2.77	1.04
boundary data collect	0.8	0.38
memcpy GPU to CPU	1.3	0.83
MPI communication	2.1	1.01
memcpy CPU to GPU	1.3	0.83
Total time(measured)	8.3	4.1
GFLOPS(measured)	19 GF	36 GF

Table 1: A single Dirac operation time table at single and double precision calculation. GFLOPS is derived from entire CG sequence, not from the 1 dirac operation.

and PCI-bus, then the pure GPU performance is 145(SP), and 55(DP) giga flops. So, in order to optimize the CG code using multi GPUs, we must focus on the data communication.

By increasing the data packet size as large as possible, we can achieve the maximum memory bandwidth in the data communication. If we can overlap cudamemcpy with MPI communication by using asynchronous communication, then we can reduce the total communication time by almost 1/3. But for this, we need to use the *GPU direct 1.0 technology*[9] and this function is only supported on *Tesla* series of GPU not on the *GeForce* series. So we could not apply this functionality in our machine yet.

Because single precision FLOPS in table.1 is less than twice of double precision case, it may look a little bit weird. But for entire program sequence, there are additional process to launch the CG operation, such as loading data in CPU memory and rearranging data for the coalesced access. Because single precision calculation can finish the job faster, entire performance in single precision is more sensitive to these extra tasks. So the actual single precision performance is slightly less than twice.

4. Future Perspective

In the version of CUDA(< 4.0), even if two or more multiple GPUs are connected to the same computing node, we need memcpy process between GPU memory and CPU memory to send the GPU data to another GPU memory. Recently, Nvidia has announced *GPU direct technology 2.0* which support direct memory copy between different GPUs within a single machine node [4]. We plan to implement this new method to our CG code in near future. However, for off-node communication, we still need memory copy process between GPU and CPU. But removing unwanted memory traffic, we can make the whole memory bandwidth dedicate to the off-node communication only. In the end, this function will bring performance enhancement on multi GPU communication.

5. Conclusion

By using GPUs, we can get a good performance in the CG algorithm for staggered fermions. The final performance is about 145(SP), 55(DP) GFLOPS per GPU(on GTX 480). We notice that

data transfer between GPU and GPU memory is a main bottle-neck. For better performance, various optimization methods are used. Including the MPI network communication, the performance is reduced down to 36(SP), 19(DP) GFLOPS per GPU. The CUDA code of CG with multi GPUs runs in the production mode to calculate hadron spectrum and weak matrix elements relevant to CP violation in the neutral kaon system [10, 11] at present.

6. Acknowledgments

The research of W. Lee is supported by the Creative Research Initiatives Program (3348-20090015) of the NRF grant funded by the Korean government (MEST).

References

- [1] Hyung-Jin Kim, Weonjong Lee, Multi GPU Performance of Conjugate Gradient Algorithm with Staggered Fermions. PoS(Lattice 2010) 028
- [2] Martin, R. and Peters, G. and Wilkinson, J., Iterative refinement of the solution of a positive definite system of equations, *Numerische Mathematik*, Vol 8, 203-216, 1966
- [3] Strzodka, Robert and Goddeke, Dominik, Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components, *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 259-270, 2006
- [4] Nvidia Corporation, “NVIDIA CUDA programming Guide”, 2010,
http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf
- [5] Sleijpen, G. L. G. and van der Vorst, H. A., “Reliable updated residuals in hybrid Bi-CG methods”, *Computing*, 2(1996), Vol 56, 141–163
- [6] M.A. Clark, R. Babich, K. Barros, R.C. Brower, C. Rebbi, “Solving lattice QCD systems of equations using mixed precision solvers on GPUs”, *Computer Physics Communication*, 182 (2010), 1517-1528.
- [7] John K. Reid, *On the Method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations, Large Sparse Sets of Linear Equations* (London and New York) (John K. Reid, ed.), Academic Press, London and New York, 1971, pp. 231-254.
- [8] Leonard Susskind, “Lattice fermions”, *Phys. Rev. D*, 16, 3031-3039 (1977).
- [9] Mellanox Technologies Corporation, “NVIDIA GPU Direct Technology-Accelerating GPU-based Systems”, http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf
- [10] Weonjong Lee, *et al.*, PoS (LATTICE 2011) 316; [arXiv:1110.2576].
- [11] Kwangwoo Kim, *et al.*, PoS (LATTICE 2011) 313; [arXiv:1110.2575].